Proceedings of the

Deduktionstreffen 2019

Meeting of the Special Interest Group on
Deduction Systems (FG DedSys)

associated with KI 2019

Kassel, Germany, September 23th, 2019

Edited by Claudia Schon and Alexander Steen

Preface

The annual meeting Deduktionstreffen is the prime activity of the Special Interest Group on Deduction Systems (FG DedSys) of the AI Section of the German Society for Informatics (GI-FBKI). It is a meeting with a familiar, friendly atmosphere, where everyone interested in deduction can report on their work in an informal setting.

A special focus of the Deduktionstreffen is on young researchers and students, who are particularly encouraged to present their ongoing research projects to a wider audience. Another goal of the meeting is to stimulate networking effects and to foster collaborative research projects.

Deduktionstreffen 2019 is associated with the German KI 2019, which brings together academic and industrial researchers from all areas of AI, providing an ideal place for exchanging news and research results of intelligent system technology. The Deduktionstreffen also hosted the annual general meeting of the members of FG DedSys.

We want to thank Konstantin Korovin (U. Manchester) for accepting our invitation to give a keynote talk about *Solving non-linear constraints in the CDCL style*. Finally, the Deduktionstreffen 2019 organizers seize the opportunity to thank the Program Committee members for their most valuable comments on the submissions, the authors for inspiring papers, the audience for their interest in this workshop, and the organizers of the KI 2019 workshop program for their support.

September, 2019 Claudia Schon
Koblenz

Table of Contents

Program Committee

Serge Autexier
Bernhard Beckert
Christoph Benzmüller
Jasmin Blanchette
Jürgen Giesl
Manfred Kerber
Jens Otten
Florian Rabe
Claudia Schon (co-chair)
Stephan Schulz
Viorica Sofronie-Stokkermans
Alexander Steen (co-chair)
Uwe Waldmann

,

Computing Expected Runtimes for Constant Probability Programs[⋆]

Jürgen Giesl[1], Peter Giesl[2], and Marcel Hark[1]

[1] LuFG Informatik 2, RWTH Aachen University, Germany
{giesl,marcel.hark}@cs.rwth-aachen.de
[2] Department of Mathematics, University of Sussex, UK
p.a.giesl@sussex.ac.uk

In recent years, probabilistic programs have gained a lot of interest. They are used to describe randomized algorithms and probability distributions, with applications in computer vision, statistical modelling, and machine learning. However, when it comes to termination analysis, probabilistic programs are conceptually harder than their deterministic counterpart. Nevertheless, for deterministic programs, special classes have been determined where termination is decidable (cf., e.g., [4]). In this work, we focus on *constant probability programs*, a special class of probabilistic programs. By using results from random walk theory, we show that for these programs there is a very simple procedure to decide the termination behavior. Moreover, we also show that the expected runtimes of constant probability programs can be computed *exactly* and present an implementation in our tool KoAT [1].

As an example, consider the well-known program which models the race between a tortoise and a hare (see, e.g., [2]). As long as the tortoise (variable t) is not behind the hare (variable h), it does one step in each iteration. With probability $\frac{1}{2}$, the hare stays at its position and with probability $\frac{1}{2}$ it does a random number of steps uniformly chosen between 0 and 10. The race ends when the hare is in front of the tortoise. Here, the hare wins with probability one and recent techniques infer the upper bound $\frac{2}{3}\cdot\max(t-h+9,0)$ on the expected number of loop iterations. Thus, the program is positively almost surely terminating.

```
while ($h \leq t$) {
  $t = t + 1$;
  $\{h = h + \mathit{Unif}(0,10)\} \oplus_{\frac{1}{2}} \{h = h\}$;
}
```

The idea of our decision procedure is the following: we first transform programs into a form with only one program variable, e.g., by only considering the distance $t-h+1$ of the tortoise and the hare. Then our procedure finds out that it is expected to decrease by $\frac{3}{2}$ in each loop iteration. We say the *drift* of the program is $-\frac{3}{2}$. This can be deduced directly from the syntax of the program by considering the changes of the distance with the according probabilities. As already mentioned, results from random walk theory yield that the sign of the drift *decides* the termination behavior for such programs: a negative drift implies positive almost sure termination, a drift of zero implies almost sure termination with infinite expected runtime and a positive drift implies that the probability of

[⋆] Supported by the DFG Research Training Group 2236 UnRAVeL and the London Mathematical Society (Grant 41662, Research in Pairs).

nontermination is strictly greater than zero. Furthermore, the drift can be used to compute upper and lower bounds on the expected runtime *directly*. In the case of the tortoise and the hare, the expected runtime is between $\frac{2}{3} \cdot (t - h + 1)$ and $\frac{2}{3} \cdot (t - h + 1) + \frac{16}{3}$, i.e., it is asymptotically linear.

However, our procedure can even compute the expected runtime of such programs *exactly*. To this end, we describe the expected runtime of a constant probability program as the *least nonnegative* solution of a linear recurrence equation. Linear recurrence equations are well studied in mathematics. The complex vector space of *all* solutions of such an equation can be computed by using the roots of the *characteristic polynomial*, and the degree of this polynomial is the dimension of the solution space. Modern computer algebra systems such as SymPy can easily compute a basis of this space. Nevertheless, determining the *least nonnegative* solution has not been considered so far and is more involved. But we have seen that the asymptotic expected runtime of a constant probability program is linear. Furthermore, in the case of the tortoise and the hare, if the distance between tortoise and hare is smaller or equal to zero, the race has finished, i.e., if this distance is smaller or equal to zero, then the expected runtime is zero. By combining these observations with the standard procedure for solving linear recurrence equations we can deduce the exact expected runtime in terms of algebraic numbers, i.e., it is possible to compute the exact expected runtime up to any chosen precision. For instance, for the race of tortoise and hare our algorithm computes

$$\begin{aligned} \mathit{runtime}(t,h) = {} & 0.049 \cdot 0.65^{(t-h+1)} \cdot \sin(2.8 \cdot (t-h+1)) - 0.35 \cdot 0.65^{(t-h+1)} \cdot \cos(2.8 \cdot (t-h+1)) \\ & +0.15 \cdot 0.66^{(t-h+1)} \cdot \sin(2.2 \cdot (t-h+1)) - 0.35 \cdot 0.66^{(t-h+1)} \cdot \cos(2.2 \cdot (t-h+1)) \\ & +0.3 \cdot 0.7^{(t-h+1)} \cdot \sin(1.5 \cdot (t-h+1)) - 0.39 \cdot 0.7^{(t-h+1)} \cdot \cos(1.5\,(t-h+1)) \\ & +0.62 \cdot 0.75^{(t-h+1)} \cdot \sin(0.83 \cdot (t-h+1)) - 0.49 \cdot 0.75^{(t-h+1)} \cdot \cos(0.83 \cdot (t-h+1)) \\ & +\tfrac{2}{3} \cdot (t-h) + 2.3 \end{aligned}$$

within 0.49 s. Here, the “sin” and “cos” appear when choosing a representation which does not involve any complex numbers. So, for example, if the tortoise starts 10 steps ahead of the hare we can expect the race to finish after 9 iterations.

The full version of this paper appears in [3].

References

1. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. ACM Trans. Program. Lang. Syst. **38**(4), 13:1–13:50 (2016), https://doi.org/10.1145/2866575
2. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Proc. CAV ’13. pp. 511–526. LNCS 8044 (2013), https://doi.org/10.1007/978-3-642-39799-8_34
3. Giesl, J., Giesl, P., Hark, M.: Computing expected runtimes for constant probability programs. In: Proc. CADE ’19. LNCS 11716 (2019), extended version with further details and proofs is available at https://arxiv.org/abs/1905.09544
4. Tiwari, A.: Termination of linear programs. In: Proc. CAV ’04. pp. 70–82. LNCS 3114 (2004), https://doi.org/10.1007/978-3-540-27813-9_6

Automation of Higher-Order Modal Logic via Semantic Embedding

Tobias Gleißner

Freie Universität Berlin, Institute of Computer Science
tobias.gleissner@fu-berlin.de

Introduction. Computer-assisted reasoning in non-classical logics is of increasing interest as its potential applications grow in numbers: Epistemic, doxastic and alethic logics in philosohpical disputes, input-output logics in legal reasoning, different temporal logics in verification processes and paraconsistent, public announcement, dynamic, deontic, multi-agent and description logics in AI contexts. All these examples employ non-classical logics that are tailored to a specific (part of a) domain. Unfortunately, with a few exceptions, most reasoning systems can process only classical logics or a (often propositional) fragment of one specific non-classical logic. As designing and implementing reasoning software is very costly, creating provers for special purpose logics usually is not a feasible course of action. The semantic embedding approach remedies this situation: By encoding the semantics of the logic of interest (source logic) in some other logic (target logic) and augmenting a hypothesis and its axioms accordingly, off-the-shelf reasoners for the target logic become applicable for solving a problem of the source logic. In this abstract higher-order modal logic [16, 10, 11] will be the source logic to exemplify the automation of a non-classical logic via the semantic embedding approach based on the work of Lewis [14] Benzmüller and Paulson [4, 6]. The target logic selected here is classical higher-order logic, which is automated by several mature reasoning systems such as Leo III [1], Satallax [8] and Nitpick [7]. Semantic embeddings for numerous other non-classical logics including conditional logics [2], hybrid logic [18], intuitionistic logics [3], free logics [5] and many-valued logics [17] have been proposed and partially implemented [15].

Automation of Higher-Order Modal Logic. The strongly simplified description of the semantics of modal logics for this showcase is as follows: There exists a set of possible worlds on which any proposition is evaluated individually. A so-called accessibility relation is defined that (partly) connects these world. A new operator $\Box$ which can be applied to propositions extends the signature of classical logics and is evaluated to true if and only if the proposition evaluates to true in all reachable worlds with respect to the accessiblity relation. A proposition is a tautology if and only if it is true on every world.
For the semantic embedding the set of worlds is encoded as a new type μ and a relation $r_{\mu\to\mu\to\sigma}$ mimicking the acessibility relation is introduced. All operators are replaced by a world-dependant variant e.g. disjunction $\vee_{o\to o\to o}$ in modal logic becomes $\lambda A_{\mu\to\sigma}, B_{\mu\to\sigma}, W_{\mu}.\ A\ W \vee B\ W$ in the target logic. The operator $\Box_{o\to o}$ is exchanged by $\lambda A_{\mu\to\sigma}, W_{\mu}.\ \forall V.\ r\ W\ V \supset A\ V$ according to its semantics that demand the proposition A to hold on all reachable worlds V from world W. Finally a proposition A can be expected to be true on every world in order to become valid by quantifying over the set of worlds and applying the worlds to the proposition similar to $\forall W_{\mu}.\ A\ W$.

Evaluation. This approach has been implemented in the Modal Embedding Tool (MET) [10, 13, 12, 11] and shown to have competitive performance when paired with state-of-the-art higher-order reasoning systems and compared against the most advanced native reasoning system [10, 11]. Furthermore non-classical logics often yield a vast amount of semantic variations as it is the case with higher-order modal logic: The $\Box$-operator can impose certain properties like transitivity on the accessibility relation, constants may denote different objects on different worlds, domains might not be assumed identical when compared world-wise, multiple accessibility relations could be defined resulting in more than one $\Box$-operator and there is more than one form of logical consquence [9, 10, 11]. These semantic variants can be easily handled and combined in a semantic embedding implementation since the modifications sum up to providing alternative definitions and adding some axioms.

Summary. The semantic embedding approach is a successful way to automate logics that do lack sophisticated reasoning software. It is cheap to build, competetive against native reasoning systems and can be quickly adapted for a huge range of semantic alternatives.

References

[1] Benzmüller, C., Paulson, L.C., Sultana, N., Theiß, F.: The higher-order prover LEO-II. Journal of Automated Reasoning 55(4), 389–404 (2015)

[2] Benzmüller, C.: Cut-Elimination for Quantified Conditional Logic. J. of Philosophical Logic (2016)

[3] Benzmüller, C., Paulson, L.: Multimodal and intuitionistic logics in simple type theory. The Logic Journal of the IGPL 18(6), 881–892 (2010)

[4] Benzmüller, C., Paulson, L.: Quantified Multimodal Logics in Simple Type Theory. Logica Universalis (Special Issue on Multimodal Logics) 7(1), 7–20 (2013)

[5] Benzmüller, C., Scott, D.: Automating free logic in Isabelle/HOL. In: Greuel, G.M., Koch, T., Paule, P., Sommese, A. (eds.) Mathematical Software – ICMS 2016, 5th International Congress, Proceedings. LNCS, vol. 9725, pp. 43–50. Springer, Berlin, Germany (2016)

[6] Benzmüller, C., Woltzenlogel Paleo, B.: Higher-order modal logics: Automation and applications. In: Paschke, A., Faber, W. (eds.) Reasoning Web 2015. pp. 32–74. No. 9203 in LNCS, Springer, Berlin, Germany (2015), (Invited paper)

[7] Blanchette, J., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Proc. of ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer (2010)

[8] Brown, C.: Satallax: An automated higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Proc. of IJCAR 2012. LNAI, vol. 7364, pp. 111 – 117. Springer (2012)

[9] Fitting, M., Mendelsohn, R.: First-Order Modal Logic. Synthese Library Studies in Epistemology Logic, Methodology, and Philosophy of Science Volume 277, Springer (1998)

[10] Gleißner, T.: Converting Higher-Order Modal Logic Problems into Classical Higher-Order Logic. Bsc. thesis, Freie Universität Berlin, Institute of Computer Science, Berlin, Germany (2016)

[11] Gleißner, T.: A Framework for Higher-Order Modal Logic Theorem Proving. Msc. thesis, Freie Universität Berlin, Institute of Computer Science, Berlin, Germany (2019)

[12] Gleißner, T., Steen, A.: The met: The art of flexible reasoning with modalities. In: Benzmüller, C., Ricca, F., Parent, X., Roman, D. (eds.) Rules and Reasoning. pp. 274–284. Springer International Publishing, Cham (2018)

[13] Gleißner, T., Steen, A., Benzmüller, C.: Theorem provers for every normal modal logic. In: Eiter, T., Sands, D. (eds.) LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 46, pp. 14–30. EasyChair, Maun, Botswana (2017), https://easychair.org/publications/paper/340346

[14] Lewis, D.K.: Counterpart theory and quantified modal logic. The Journal of Philosophy 65(5), 113–126 (1968), http://www.jstor.org/stable/2024555

[15] Makarenko, I.: Automatisierung von freier Logik in Logik höherer Stufe. Bsc. thesis, Freie Universität Berlin, Institute of Computer Science, Berlin, Germany (2016)

[16] Muskens, R.: Higher order modal logic. Handbook of modal logic 3 (2007)

[17] Steen, A., Benzmüller, C.: Sweet SIXTEEN: Automation via embedding into classical higher-order logic. Logic and Logical Philosophy 25, 535–554 (2016)

[18] Wisniewski, M., Steen, A.: Embedding of Quantified Higher-Order Nominal Modal Logic into Classical Higher-Order Logic. In: Benzmüller, C., Otten, J. (eds.) Automated Reasoning in Quantified Non-Classical Logics (ARQNL), Proceedings. EPiC, vol. 33, pp. 59–64. EasyChair (2014)

Automatic Modularization of Large Programs for Bounded Model Checking

Marko Kleine Büning

Karlsruhe Institute of Technology (KIT), Germany
marko.kleinebuening@kit.edu

1 Extended Abstract

The verification of real-world applications is a continuous challenge which yielded numerous different methods and approaches. However, scalability of precise analysis methods on large programs is still limited. One of the reasons is the growth in size of embedded systems. Modern cars are currently at around 100 MLoC and are estimated to go up to a total of 300 MLoC in the next years.

Problem statement. For bounded model checking (BMC), a program under verification has to be encoded into a logical formula. Even when ignoring time constraints, the memory requirements to encode millions of lines of code is not attainable by state-of-the-art BMC systems. A well-known approach to increase scalability of software verification is to partition the program into smaller modules that can then be solved individually [2,5]. Such modularization typically requires formalization of interfaces and dependencies between modules. Current work generally does not cover the aspect of how to generate modules. There exist frameworks that automate part of the modularization task, e.g. by precondition learning or deduction of modules from program design [3,4]. However, these approaches do not provide a framework for fully automatic verification of large systems. Tools using separation logic for modularization are among others ETH's Viper [7] and Facebook's INFER [1], which are loosely related to our approach but concentrate either on manual specification or limited memory properties.

Proposed solution. The presented work is an extract of the content published in [6]. To automatically verify large projects, an automatic modularization is needed. Therefore, we first created formal definitions of program semantics and modularization. Based on these definitions, we developed four modularization approaches which are based on abstractions. Abstractions are an important technique to simplify verification tasks. Most often abstractions are over-approximations of variable values. The abstractions that we are interested in are different and of a “structural” kind. We abstract function calls and replace them by over-approximations of the function behavior, or we ignore the calling context of a function in a larger program.

We illustrate the modularization approaches in Fig. 1. Every node represents a function while the edges represent function calls. The triangles represent property checks that are inserted into a function. The green boxes designate the module and the green arrow marks the entry point for the analysis.

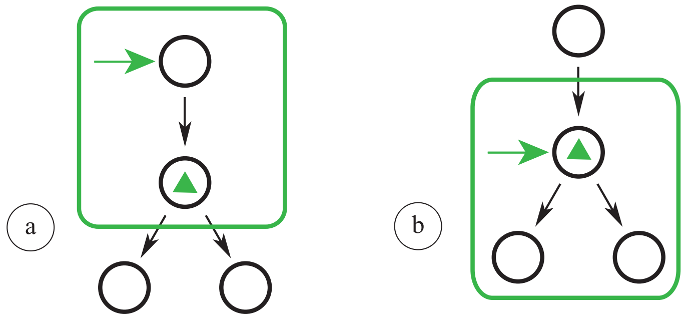


Fig. 1: Illustration of the modularization approaches:
(a) Function behavior abstraction; (b) calling context abstraction.

The first two approaches over-approximate the function behavior of called functions as e. g. shown in Fig. 1(a). The analysis starts at the main function to generate a precise calling context for the function to be analyzed. For the abstraction of the called functions, we have two possibilities:

1. **Havoc function call.** Without any further knowledge about the function behavior, we have to assume arbitrary values for the return value and all memory content.
2. **Postcondition generation.** We implement a prepossessing step that generates postconditions for the function behavior of havoced functions. In a first step that includes only a set of changed memory locations and is then extended to value calculations.

The third and the fourth approach abstract the calling context of a function as e. g. shown in Fig. 1(b). We abstract all prior function calls but therefore include all functions called, i.e. no havocing. There are again two possibilities to abstract the calling context of a function:

1. **"Library" abstraction.** Without any further knowledge, we set the memory and all input parameters to arbitrary values at the start of the function. This abstraction is particular useful when analyzing libraries or functions accessible throughout the system.
2. **Precondition generation.** We can generate preconditions for our function. To calculate exhaustive preconditions for large programs is currently not feasible. Thus, we generate them bottom up based on erroneous checks.

For evaluation, we compared a whole-program analysis approach to the library abstraction and showed that for industry sized projects with ca. 160KLoC, the modularization approach heavily outperforms the global analysis. To further improve automatic verification of large programs, current and future work includes the detailed development and implementation of the remaining modularization approaches. The automatic creation of pre- and postconditions will likely reduce the amount of potential false positives. Additionally, an customized alias analysis is developed to argue and refine data dependencies of programs. Automatic verification of industry sized applications is still a huge challenge that will accompany any sound verification approach. We see the automatic modularization as a promising concept to handle such programs. To fully utilize the advantages of modularization, systems can be adjusted such that every module analysis can be run in a parallel setting.

References

1. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods. pp. 3–11. Springer International Publishing, Cham (2015)
2. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science. pp. 353–362. IEEE (1989)

3. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 331–346. Springer (2003)
4. Giannakopoulou, D., Pasareanu, C.S., Cobleigh, J.M.: Assume-guarantee verification of source code with design-level assumptions. In: Proceedings. 26th International Conference on Software Engineering. pp. 211–220. IEEE (2004)
5. Grumberg, O., Long, D.E.: Model checking and modular verification. In: International Conference on Concurrency Theory. pp. 250–265. Springer (1991)
6. Kleine Büning, M., Sinz, C.: Automatic modularization of large programs for bounded model checking. In: Formal Methods and Software Engineering - 21th International Conference on Formal Engineering Methods, ICFEM 2019, November 5th-9th, Proceedings (2019), accepted and to be published
7. Müller, P.: Modular specification and verification of object-oriented programs. Springer-Verlag (2002)

Automated Machine Learning Based Software Quality Assurance

Safa Omri

Karlsruhe Institute of Technology (KIT), Germany
safa.omri@kit.edu

1 Introduction

Software quality assurance is overall the most expensive activity in safety-critical embedded software development. Increasing the effectiveness and efficiency of software quality assurance is more and more important given the size, complexity, time and cost pressures in automotive development projects. Therefore, in order to increase the effectiveness and efficiency of software quality assurance tasks, we need to identify problematic code areas most likely to contain program faults and focus the quality assurance tasks on such code areas. Obtaining early estimates of fault-proneness helps making decisions on testing, code inspections and design rework, to financially plan possible delayed releases, and affordably guide corrective actions to the quality of the software.

One source to identify fault-prone code components can be their failure history, which can be obtained from bug databases; a software component likely to fail in the past is likely to do so in the future. However, in order to get accurate predictions, a long failure history is required. Such a long failure history is usually not available, moreover maintaining long failure histories is usually avoided altogether. A second source to estimate fault-proneness of software components is the program code itself. Static code analysis and code complexity metrics have been shown to correlate with fault density in a number of case studies. Furthermore, faults are usually strongly related to changes made in the software systems and learning the changes which occur throughout software evolution by code churn is additionally essential. Different recent works have used past changes as indicators for faults.

Our major question is whether or not we can use code complexity metrics combined with static analysis fault density and with code churn metrics to predict pre-release fault density, that is: Is combining static analysis tools with code complexity metrics and with code churn metrics a leading indicator of faulty code?

We present an exploratory study investigating whether the faults detected by static analysis tools combined with code complexity metrics and with code churn metrics can be used as software quality indicators and to build pre-release fault prediction models. The combination of code complexity metrics with static analysis fault density and with code churn metrics was used to predict the pre-release fault density with an accuracy of 78.3%. This combination can also be used to separate high and low quality components with a classification accuracy of 79%.

In order to enhance the prediction accuracy of our old prediction approach, we believe that the syntax and different levels of semantics of the studied source code have to be also considered as an independent variables of the prediction process.

2 Study Design

Our experiments were carried out using eight software projects of an automotive head unit control system (Audio, Navigation, Phone, etc.). Each project, in turn, is composed of a set of components. The total number of components is 54. These components have a collective size of 23.797 MLOC (million LOCs without comments and spaces). All components use the object oriented language C++. For each of the components, we compute a number of metrics, as described in Table 1.

3 Case Study

Our process can be summarized in the following three steps:

Table 1. Metrics used for the study

Metrics	**Description**
Static Analysis Fault Density	# faults found by static analysis tools per KLOC (thousand lines of code).
Code Churn Metrics	
Relevant LOC	# relevant LOCs without comments, blanks, expansions, etc.
Added LOC	# lines of code added
Removed LOC	# lines of code deleted
Modified Files	# files modified
Developpers	# developers
Code Complexity Metrics	
Complexity	cyclomatic complexity of a method
Nesting	# nesting levels in a method
Statements	# statements in a method
Paths	# non-cyclic paths in a method
Parameters	# function parameters in a method

3.1 Data Preperation

The data required to build our fault predictors are the metrics described in in Table 1, and the Pre-release faults where we mine the archives of several major software systems at Daimler and map their pre-release faults (faults detected during development) back to its individual components. We define the pre-release fault density of a software component as the number of faults per KLOC found by other methods (e.g. testing) before the release of the component.

3.2 Model Training

We train statistical models to learn the pre-release fault densities based on a) static analysis faults densities, b) code complexity metrics, and c) code churn metrics.

3.3 Model Prediction

The trained statistical models are used to

a) predict pre-release fault densities of software components (Regression): we applied statistical regression techniques where the dependent variable is the pre-release fault density, and the independent variables are the code complexity metrics combined with the static analysis fault density. The models we tested include linear, exponential, polynomial regression models as well as support vector regressions and random forest.
b) discriminate fault-prone software components from the not fault-prone software components (Classification): we applied several statistical classification techniques. The classification techniques include random forest classifiers, logistic regression, passive aggressive classifiers, gradient boosting classifiers, K-neighbors classifiers and support vector classifiers.

4 Conclusion and Future Work

In this work, we verified the following hypotheses: (1) static analysis fault density combined with code complexity metrics are good predictor of pre-release fault density and a good discriminator between fault-prone and not fault-prone components; and (2) the history of code changes (code churn metrics) between different commits and releases when combined with (1) improves the prediction accuracy of the pre-release faults.

We plan to further validate our study by evaluating the semantic and syntax of the source code using the abstract syntax tree (AST) and control- and dataflow as source code representation. We also plan to train deep learning models to predict software faults not only on Component level but also on the method level.

Systematic Analysis of Experiments in Solving Boolean Satisfiability Problems

Markus Iser

Karlsruhe Institute of Technology (KIT)
markus.iser@kit.edu

Introduction. Aside from its theoretical weight, the SAT problem is of interest in numerous practical domains like software and hardware verification [7, 6], AI planning [12] and scheduling [1] or electronic design automation [10].

Successful recent SAT solvers (i.e. Conflict-Driven Clause-Learning (CDCL) solvers) exploit problem structure [2]. Structure features of SAT problems come in many forms and can be based on literal occurrences [13], problem partitions [5], graph representations [3] or combinatorial circuit encodings [8].

As some strategies and configurations work well on specific domains of problems but not so on others, portfolio approaches like SATzilla [14] or ISAC [3] use machine learning to estimate the best algorithm and configuration for a problem. Such approaches might work well in practice but come with a lack of scientific insight.

Systematic approaches that analyze why and when heuristics work and how much of an impact these heuristics have are rare [4]. In [9], Katebi et al. present an attempt to analyze the impact of several state-of-the-art strategies including an evaluation that distinguishes problems by family.

We present a systematic approach in order to answer to the following questions. Are there “lost” approaches in SAT solving (consider [11]) which are only successful on a specific problem domain but inferior in the overall picture? Why do some heuristics work best on certain domains? Why are certain configurations and combinations of heuristics more successful than others?

Approach. We are working on two software projects which allow us to systematically analyze competing strategies in SAT solving. This includes the development of the modular SAT solver Candy[1] as well as the benchmark feature database Global Benchmark Database (GBD)[2] which allows us to analyze the results of runtime experiments in combination with benchmark features.

Candy. Candy is a modular SAT solver consisting of a set of exchangeable *solver systems*, e.g. the propagation system, the restart system, the clause-learning or the branching system. Candy includes systems that implement the common CDCL algorithms and data-structures, but provides also e.g. alternative branching strategies [8]. All systems share common references to the objects managing the *clause database* and the *current assignment*, thus enabling cross-system communication. Lightweight system interfaces simplify the implementation of new strategies.

[1] https://github.com/Udopia/candy-kingdom
[2] https://github.com/Udopia/gbd

GBD. Our benchmark database tool allows to manage and distribute benchmark meta-data and comes with a command-line interface as well as a web-frontend. GBD allows to query for benchmark features and to download benchmark problems based on the selection of specific features. The command-line tool allows to store benchmark features as well as solver runtimes in the database.

Conclusion. Both systems in combination allow for a systematic analysis of old and new strategies which are used in SAT solvers. As runtime and feature data becomes archived, it can be revisited anytime. The development of new solver strategies for Candy or new machine learning approaches for GBD can now also be integrated into small scope student projects.

References

1. Alves, R., Alvelos, F., Sousa, S.D.: Resource constrained project scheduling with general precedence relations optimized with SAT. In: Progress in Artificial Intelligence, pp. 199–210. Springer (2013)
2. Ansótegui, C., Bonet, M.L., Giráldez-Cru, J., Levy, J.: Community structure in industrial SAT instances. CoRR (2016)
3. Ansótegui, C., Bonet, M.L., Giráldez-Cru, J., Levy, J.: Structure features for SAT instances classification. Journal of Applied Logic **23**, 27–39 (Sep 2017)
4. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence. pp. 399–404. IJCAI'09 (2009)
5. Biere, A., Sinz, C.: Decomposing SAT problems into connected components. JSAT **2**(1-4), 201–208 (2006)
6. Biere, H., Järvisalo, M.: Equivalence checking of HWMCC 2012 circuits. In: Proceedings of SAT Competition 2013. p. 104 (2013)
7. Falke, S., Merz, F., Sinz, C.: LLBMC: Improved bounded model checking of C programs using LLVM - (competition contribution). In: TACAS. pp. 623–626 (2013)
8. Iser, M., Kutzner, F., Sinz, C.: Using gate recognition and random simulation for under-approximation and optimized branching in SAT solvers. In: ICTAI (2017)
9. Katebi, H., Sakallah, K.A., Marques-Silva, J.P.: Empirical study of the anatomy of modern sat solvers pp. 343–356 (2011)
10. Mihal, A., Teig, S.: A constraint satisfaction approach for programmable logic detailed placement. In: SAT. pp. 208–223 (2013)
11. Nadel, A., Ryvchin, V.: Chronological backtracking. In: Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. pp. 111–121 (2018)
12. Rintanen, J.: Engineering efficient planners with SAT. In: ECAI. pp. 684–689 (2012)
13. Silva, J.P.M.: The impact of branching heuristics in propositional satisfiability algorithms. In: Proc. Portuguese Conference on Artificial Intelligence EPIA. pp. 62–74 (1999)
14. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based Algorithm Selection for SAT. Journal of Artificial Intelligence Research **32**, 565–606 (Jul 2008)

Automated Reasoning with Complex Ethical Theories
— A Case Study Towards Responsible AI —

David Fuenmayor[1] and Christoph Benzmüller[1,2]

[1] Dep. of Mathematics and Computer Science, Freie Universität Berlin, Germany
[2] Faculty of Science, Technology and Communication, University of Luxembourg, Luxembourg

The design of *explicit* ethical agents [8] is faced with tough philosophical and practical challenges. We address in this work one of its biggest ones: How to *explicitly* represent ethical knowledge and use it to carry out complex reasoning with incomplete and inconsistent information in a scrutable and auditable fashion, i.e. interpretable for both humans and machines. We present a case study illustrating the utilization of higher-order automated reasoning for the representation and evaluation of a complex ethical argument, using a Dyadic Deontic Logic (DDL) [3] enhanced with a 2D-Semantics [6]. This logic (DDL) is immune to known paradoxes in deontic logic, in particular "contrary-to-duty" scenarios. Moreover, conditional obligations in DDL are of a defeasible and paraconsistent nature and so lend themselves to reasoning with incomplete and inconsistent data.

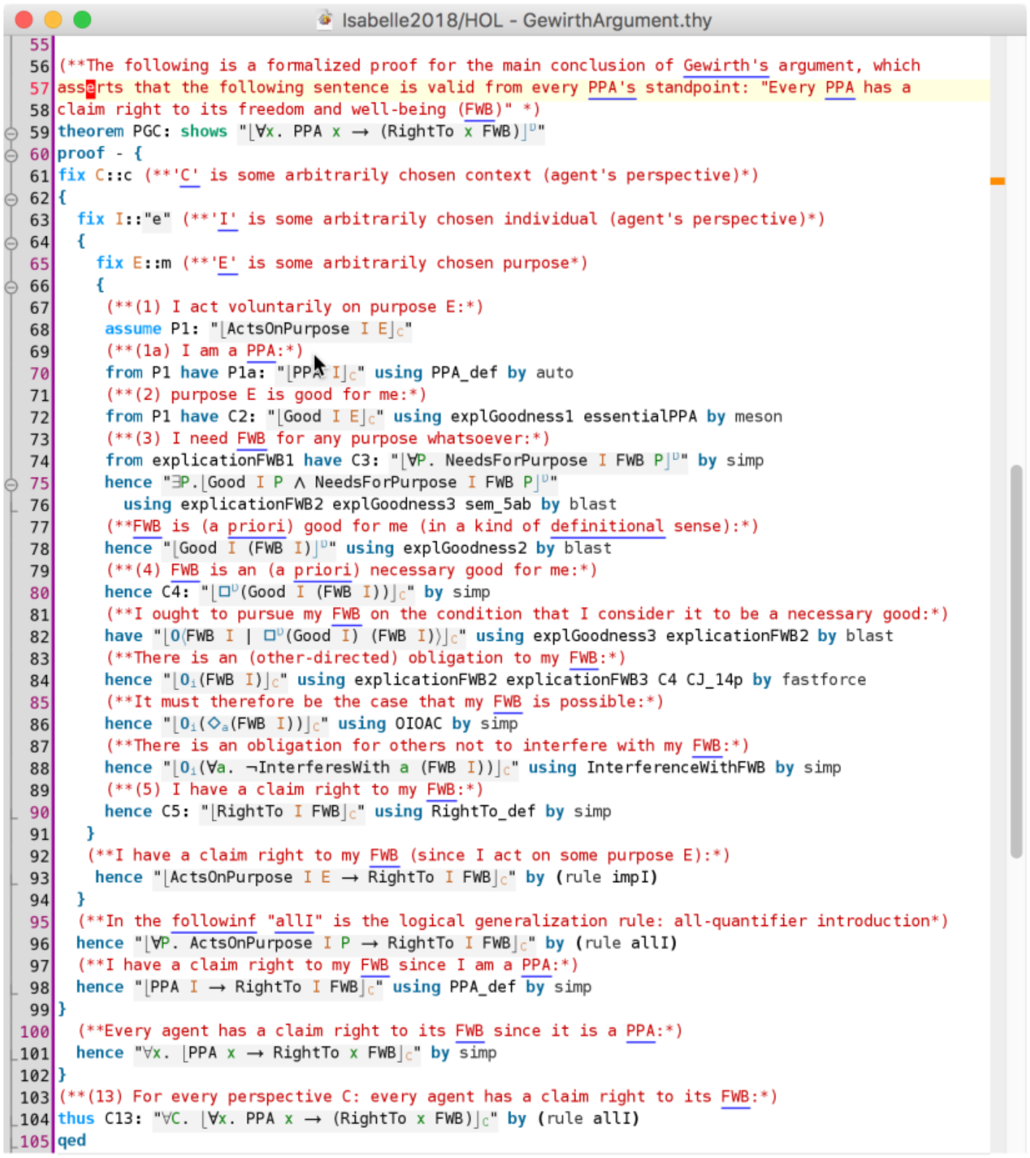
Our case study consists of a rational argument originally presented by the philosopher Alan Gewirth [5], which aims at justifying an upper moral principle: the "Principle of Generic Consistency" (PGC). It states that any agent (by virtue of its self-understanding as an agent) is rationally committed to asserting that (i) it has rights to freedom and well-being, and that (ii) all other agents have those same rights. The argument used to derive the PGC is by no means trivial and has stirred much controversy in legal and moral philosophy during the last decades and has also been discussed as an argument for the a priori necessity of human rights. Most interestingly, the PGC has lately been proposed as a means to bound the impact of artificial general intelligence (AGI) by András Kornai [7]. Kornai's proposal draws on the PGC as the upper ethical principle which, assuming it can be reliably represented in a machine, will guarantee that an AGI respects basic human rights (in particular to freedom and well-being), on the assumption that it is able to recognize itself, as well as humans, as agents capable of acting voluntarily on self-chosen purposes.

In the annexed Fig. 1 we show an extract of our work on the formal reconstruction of Gewirth's argument for the PGC using the proof assistant Isabelle/HOL (a formally-verified, unabridged version is available in the Archive of Formal Proofs [4]). Independent of Kornai's claim, our work exemplarily demonstrates that reasoning with ambitious ethical theories can meanwhile be successfully automated. In particular, we illustrate how it is possible to exploit the high expressiveness of classical higher-order logic as a metalanguage in order to embed the syntax and semantics of some object logics (e.g. DDL enhanced with quantification and contextual information) thus turning a higher-order prover into an universal reasoning engine [1] and allowing for seamlessly combining and reasoning about and within different logics (modal, deontic, epistemic, etc.).

References

1. C. Benzmüller. Universal (meta-)logical reasoning: Recent successes. *Science of Computer Programming*, 172:48–62, March 2019.
2. C. Benzmüller, X. Parent, and L. van der Torre. A deontic logic reasoning infrastructure. In F. Manea, R. G. Miller, and D. Nowotka, editors, *14th Conference on Computability in Europe, CiE 2018, Proceedings*, volume 10936 of *LNCS*, pages 60–69. Springer, 2018.
3. J. Carmo and A. J. Jones. Deontic logic and contrary-to-duties. In *Handbook of Philosophical Logic*, pages 265–343. Springer, 2002.
4. D. Fuenmayor and C. Benzmüller. Formalisation and evaluation of Alan Gewirth's proof for the principle of generic consistency in Isabelle/HOL. *Archive of Formal Proofs*, 2018.
5. A. Gewirth. *Reason and morality.* University of Chicago Press, 1981.
6. D. Kaplan. On the logic of demonstratives. *Journal of Philosophical Logic*, 8(1):81–98, 1979.
7. A. Kornai. Bounding the impact of AGI. *Journal of Experimental & Theoretical Artificial Intelligence*, 26(3):417–438, 2014.
8. J. Moor. Four kinds of ethical robots. *Philosophy Now*, 72:12–14, 2009.

Appendix



Isabelle2018/HOL - GewirthArgument.thy

```

(**The following is a formalized proof for the main conclusion of Gewirth's argument, which
asserts that the following sentence is valid from every PPA's standpoint: "Every PPA has a
claim right to its freedom and well-being (FWB)" *)
theorem PGC: shows "⌊∀x. PPA x → (RightTo x FWB)⌋D"
proof - {
fix C::c (**'C' is some arbitrarily chosen context (agent's perspective)*)
{
  fix I::"e" (**'I' is some arbitrarily chosen individual (agent's perspective)*)
  {
    fix E::m (**'E' is some arbitrarily chosen purpose*)
    {
     (**(1) I act voluntarily on purpose E:*)
     assume P1: "⌊ActsOnPurpose I E⌋c"
     (**(1a) I am a PPA:*)
     from P1 have P1a: "⌊PPA I⌋c" using PPA_def by auto
     (**(2) purpose E is good for me:*)
     from P1 have C2: "⌊Good I E⌋c" using explGoodness1 essentialPPA by meson
     (**(3) I need FWB for any purpose whatsoever:*)
     from explicationFWB1 have C3: "⌊∀P. NeedsForPurpose I FWB P⌋D" by simp
     hence "∃P.⌊Good I P ∧ NeedsForPurpose I FWB P⌋D"
       using explicationFWB2 explGoodness3 sem_5ab by blast
     (**FWB is (a priori) good for me (in a kind of definitional sense):*)
     hence "⌊Good I (FWB I)⌋D" using explGoodness2 by blast
     (**(4) FWB is an (a priori) necessary good for me:*)
     hence C4: "⌊□D(Good I (FWB I))⌋c" by simp
     (**I ought to pursue my FWB on the condition that I consider it to be a necessary good:*)
     have "⌊O⟨FWB I | □D(Good I) (FWB I)⟩⌋c" using explGoodness3 explicationFWB2 by blast
     (**There is an (other-directed) obligation to my FWB:*)
     hence "⌊Oi(FWB I)⌋c" using explicationFWB2 explicationFWB3 C4 CJ_14p by fastforce
     (**It must therefore be the case that my FWB is possible:*)
     hence "⌊Oi(◇a(FWB I))⌋c" using OIOAC by simp
     (**There is an obligation for others not to interfere with my FWB:*)
     hence "⌊Oi(∀a. ¬InterferesWith a (FWB I))⌋c" using InterferenceWithFWB by simp
     (**(5) I have a claim right to my FWB:*)
     hence C5: "⌊RightTo I FWB⌋c" using RightTo_def by simp
   }
   (**I have a claim right to my FWB (since I act on some purpose E):*)
   hence "⌊ActsOnPurpose I E → RightTo I FWB⌋c" by (rule impI)
  }
  (**In the followinf "allI" is the logical generalization rule: all-quantifier introduction*)
  hence "⌊∀P. ActsOnPurpose I P → RightTo I FWB⌋c" by (rule allI)
  (**I have a claim right to my FWB since I am a PPA:*)
  hence "⌊PPA I → RightTo I FWB⌋c" using PPA_def by simp
}
  (**Every agent has a claim right to its FWB since it is a PPA:*)
  hence "∀x. ⌊PPA x → RightTo x FWB⌋c" by simp
}
(**(13) For every perspective C: every agent has a claim right to its FWB:*)
thus C13: "∀C. ⌊∀x. PPA x → (RightTo x FWB)⌋c" by (rule allI)
qed
```

Fig. 1. Representation of a variant of Gewirth's proof in Isabelle/HOL

A Parallel SAT Solver

In SAT solvers conflict clauses are generated through the Conflict Driven Clause Learning technique. When a clause could not be satisfied because of a conflict in the assignment, a new clause, explicitly excluding the conflicting assignment, is added to the initial SAT instance [2].
In parallel SAT solvers, the conflict clauses are exchanged between the different instances. If all conflict clauses were exchanged between the instances, there would occur too much overhead. So, there are different possibilities to choose specific conflict clauses to exchange.

In this paper a new way of rating conflict clauses is established. At first, a Naive Bayes Classifier is trained by using different features. The trained Naive Bayes classifier estimates the clause quality and the ”good” conflict clauses will be shared between the instances of the SAT solver. To the best of our knowledge, algorithms from the machine learning domain have not yet been used in this context.
The parallel SAT solver is based on an existing SAT solver and exchanges only specific clauses, called ”good” clauses. To find out the ”good” conflict clauses, the Naive Bayes classifier estimates every conflict clauses quality during run time. In general, Classification is a method for determining the class a object belongs to based on several independent variables. The idea of using machine learning is based on the hope that a classifier can better evaluate conflict clauses than algorithms written by hand.

Before the Naive Bayes Classifier can be used, it has to be trained. A classifier utilizes some training data to learn how given input data relate to a class. Both for training and for actually classifying the data, features are used. While training, the conflict clause must first be assigned to classes. For each clause, the values for each feature are collected. To determine, whether the clause is a ”good” conflict clause or not, a heuristic is needed. Otherwise the classifier does not know, how to relative given input data to a class. In our approach, all conflict clauses are sorted by their activity, which is a value for each clause to record how often it was involved in a conflict. So, conflict clauses with a higher activity have lead to more conflicts. The top half of all conflict clauses are considered as ”good” clauses. With these labeled conflict clauses, the Naive Bayes classifier creates a Naive Bayes Model. From this point the classifier can be used.

This parallel SAT solver executes formulas with the competitive approach. So, multiple instances of a sequential SAT solver are executed with different initial parameters. Each instance is working on the whole formula [1].
The parameters for different search strategies will be generated randomly within a specific range. Due to the different strategies, each instance of the parallel SAT solver is searching in different areas of the search space and will generate different conflict clauses.

The basis for evaluating conflict clauses are the features. There are only values between 0 and 1 possible. The mentioned features are characteristics of a clause and they are discussed in the following.

Horn: Determines, whether the clause has at most one positive literal. Horn formulas can be solved in polynomial time and therefore it is beneficial to add horn clauses.
PosLits: Determines, whether a clause has positive literals.
NegLits: Determines, whether a clause has negative literals.
Size: The size of the clause. Small clauses are preferred, because they can be used more often during Unit Propagation. The best size of a clause is a unit clause, because we know, that this literal has to be true and other clauses which contain the unit clause can be simplified.
Distance: Determines the distance between literals indices. It is the distance between the indices of the variables in one clause. In practical encodings, variables have similar indices when they are closely related [4].
LBD: The literal block distance of a clause. Variables have a decision level and the LBD is the amount of different levels in a clause. All literals of the same level are blocks and are connected with immediate dependencies. When the solver stays in the same search space, such a clause helps to reduce the amount of decisions in the residual search. Unit Propagation of conflict clauses are based up to 40% of small LBD-values [5].
VRV: The variable rate value. It measures how often a variable gets a new assignment. Often selected variables have a higher activity and they seem to be important. Another reason for this metric is the fact that variables, assigned by the branching heuristic, should appear in more clauses, so that they even can be sometimes assigned by Unit Propagation. For this feature higher values are better than lower values.

The Naive Bayes classifier is used to assign probabilities to classes given an input object described by its features. Learning the classifier is simplified by assuming that features are independent given class. The Naive Bayes Classifier calculates the probability of the class ”good” for all features, that is

$$P(C_1 \mid f_1, .., f_7) = \frac{\prod_{i=1}^{7} P(f_i \mid C_1) \cdot P(C_1)}{\prod_{i=1}^{7} P(f_i)},$$

where $f_1, ..., f_7$ is a feature vector and C_1 is the ”good” class. The Bayes' Theorem allows to compute the probability of class $C1$ given $f_1, ..., f_7$ [3]. The value below the fraction bar can be ignored, because it is a constant and scales both posteriors equally. It has no effect to the classification. The other two values were learned by relative frequency. The probability $P(f_i|C_1)$ is modeled as a Gaussian distribution. The sum of the two resulting probabilities is 1. So, a clause could have the result of 0.4 for a "good" clause and 0.6 for a "bad" clause.

The features can be used to determine the likelihood of the conflict clause being a "good" conflict clause. The Naive Bayes classifier will return a likelihood and all clauses above a certain likelihood threshold are then passed to the other solvers. These clauses are "good" conflict clauses. It is important to choose a good threshold for a "good" conflict clause. If the likelihood is too high, only few clauses will be shared among the instances. In contrast to a too low threshold - there are too many conflict clauses and the search procedure will slowed down. Clauses which are classified as "bad", will not be forwarded, but the solver, which finds the clause still keeps them. The other instances will not interrupt their work if a thread shares a new conflict clause. They will request the "good" conflict clauses during backtracking. In consideration of the new conflict clauses, the instances try a new assignment.

The parallel SAT solver based on the existing SAT solver "MiniSat" and is written in C++. For the Naive Bayes Classifier the C++ machine learning library "mlpack" is used [6].

Ideally, there should be a correlation between the features and classes. The trained Naive Bayes model should show the same results as the statements in the literature. For example, a small conflict clause should ideally be classified as a "good" conflict clause. The features are dealing with continuous data, so the continuous values associated with each class are distributed according to a Gaussian distribution. So, the variance and the mean for each feature can be calculated. The Gaussian curve can be evaluated for the "good" conflict clauses and the "bad" conflict clauses for each feature. Unfortunately, the results do not match the findings of other researchers. For a small conflict clause there is a higher probability to be sorted into the "bad" class. The reason for this problem is probably the initial assignment of conflict clauses to "good" conflict clauses or "bad" ones based on the activity. So, another measurement is necessary here.

Nevertheless the first evaluation results are promising with the proposed approach scaling well and even outperforming the established ManySAT solver in many cases. Other benchmarks however resulted in high running times. This poses several open research questions:

- Which characteristics of the benchmarks influence the solvers running time? Is it possible to develop a perfectly trained Naive Bayes Classifier for not only a specific class of problems, but also multiple kinds of problems?
- How much overhead does the classifier generate? The classification of the conflict clauses should not take considerably more time in comparison to the rating of conflict clauses due to a specific algorithm or a heuristic.
- What is the best way to train the Naive Bayes classifier? Do we need all of the features and how can we rate them?
- Is there another good measure for the quality of a clause? There might be heuristics that describe a clauses influence on solving time better.
- How can we improve the approach?
- How can we evaluate features individually? Is it important to know positive or negative literals of a clause?
- How can we find specific problem classes, which this approach performs well on?

References

[1] S. Hölldobler and N. Manthey and Van Hau Nguyen and J. Stecklina and P. Steinke, "A short overview on modern parallel SAT-solvers," International Conference on Advanced Computer Science and Information Systems, 2011.

[2] N. Manthey, 'Parallel SAT Solving - Using More Cores', 2011.

[3] Velev, Miroslav N., "Automatic Abstraction of Equations in a Logic of Equality," International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, 2003.

[4] B., Magnus, "Successful SAT encoding techniques," Journal on Satisfiability, Boolean Modeling and Computation, 2009.

[5] G. Audemard and L. Simon, "Predicting Learnt Clauses Quality in Modern SAT Solvers*," Twenty-First International Joint Conference on Artificial Intelligence, 2009.

[6] R. Curtin and et al., 'mlpack', 2007, [Online], Available: http://www.mlpack.org/doc/mlpack-3.1.1/cli_documentation.html#nbc, [Accessed: 03.07.2019].

Computer-supported Exploration of a Categorical Axiomatization of Miroslav Benda's Modeloids

Lucca Tiemens Dana S. Scott Miroslav Benda
Christoph Benzmüller

August 5, 2019

Extended Abstract

A modeloid, formulated by Miroslav Benda[1], is an equivalence relation on the set of all words, which are sequences of non-repeating elements from a finite set, enriched with three additional axioms. The first axiom states that only words of the same length can be in the same equivalence class. The second says that if two words are equivalent, so are their sub-words of same length and the final one requires that, given two equivalent words, the permuted words stay in relation as long as they were permuted in the same way. Furthermore, a modeloid features an operation called the derivative which is inspired by Ehrenfeucht-Fraïssé games[1][4][3].

It is shown that this formulation can be generalized to category theory. This is achieved by taking advantage of the fact that a modeloid can be represented as a set of partial bijections where two words in relation become domain and image of such a partial bijection. A generalization of this set can be axiomatized as a subset of an inverse semigroup with the help of the natural partial order because the nature of being a sub-word in the setting of partial bijections can be captured by it. The Wagner-Preston representation theorem [5] assures that this generalization is faithful.

The connection to category theory is made by the natural transition from an inverse semigroup to an inverse category[7]. Since the natural partial order can be reformulated, this allows a categorical description of a modeloid. Here the derivative is shown to be capable of producing an Ehrenfeucht-Fraïssé game on the category of a finite vocabulary.

[1]See for example [6] for more information on Ehrenfeucht-Fraïssé games.

During the whole process computer-based theorem proving is employed. An inverse semigroup and an inverse category are being implemented in Isabelle/HOL[2]. Almost all proofs of inverse semigroup theory needed for the Wagner-Preston representation theorem could be found by automated theorem proving, though sometimes needing more lemmas then the amount that would be found in a textbook (e.g. in [5]). During the research on how to formulate a modeloid in category theory (and here it should be noted that a rigorous axiomatization of a category was used [2]) interactive theorem proving has helped to find the correct formulations in this setting.

References

[1] Miroslav Benda. "Modeloids. I". In: *Transactions of the American Mathematical Society* 250 (1979), pp. 47–90. DOI: `10.1090/s0002-9947-1979-0530044-4`.

[2] Christoph Benzmüller and Dana S. Scott. "Automating Free Logic in HOL, with an Experimental Application in Category Theory". In: *Journal of Automated Reasoning* (2019). Url (preprint): `http://doi.org/10.13140/RG.2.2.11432.83202`. DOI: `10.1007/s10817-018-09507-7`.

[3] Andrzej Ehrenfeucht. "An application of games to the completeness problem for formalized theories". In: *Fundamenta Mathematicae* 49.129-141 (1961), p. 13.

[4] Roland Fraïssé. "Sur quelques classifications des systèmes de relations". PhD thesis. University of Paris, 1953.

[5] Mark V Lawson. *Inverse Semigroups*. WORLD SCIENTIFIC, Nov. 1998. DOI: `10.1142/3645`.

[6] Leonid Libkin. *Elements of finite model theory*. Springer Science & Business Media, 2013.

[7] Markus Linckelmann. "On inverse categories and transfer in cohomology". In: *Proceedings of the Edinburgh Mathematical Society* 56.1 (Dec. 2012), pp. 187–210. DOI: `10.1017/s0013091512000211`. URL: `https://doi.org/10.1017/s0013091512000211`.

[2]Isabelle/HOL is a higher-order logic theorem proving environment. More information can be found at `https://isabelle.in.tum.de/index.html`

A Generic Scheduler for Large Theories presented in Batches (LTB)

Marco Träger, marco.traeger@gmail.com, Freie Universität Berlin

Reasoning applications often rely on queries with a common large set of axioms such as Suggested Upper Merged Ontology (SUMO) or Mizar Problems for Theorem Proving (MPTP) [UV13, PSST10]. To benchmark ATP systems on such problems the *Large Theories presented in Batches* (LTB) division of the CADE ATP System Competition (CASC) was created in 2008 [Sut09]. Over the years the developer of ATP systems have tested various approches to tackle TLB's. For instance: iProver using an abstraction-refinement framework [HK17] and Vampire axiom selection [KV13]. In LTB one overall time for all problems are given and the axioms are often common to the set of conjectures. Hence, one may use the results of one prove-attempt to fine tune the timeout or axiome selection for the prove-attempt of the next problem.

A Generic Scheduler for LTB I will present a ATP-agnostic Python implementation for parsing and executing LTB definitions using a modifiable scheduler approch. The implementation is not an ATP in itself, instead it is using other ATPs such as Leo III for HOL [SB18] (allowing to use specialized ATPs and a decoubling from the prove itself). The user only needs to implement a problem scheduler by implementing several callbacks which are called whenever an ATP is terminating or runs into a timeout. The user needs to provide:

- how may ATPs should be run in parallel
- which problem(s) the scheduler should start next and the time available for the attempt, proving which problem was started and ended to this time
- which ATP and ATP-specific parameter the scheduler schould use for each prove attempt

Additionally, the implementation is

- designed using efficent multithreading to be able to use as many CPU time/processors as possible
- able to kill a APT processes if the result is not longer needed
- effiently synchronized s.t. the user only implements in the main thread allowing simple code but fast execution under the hood
- generating profiling data and plots of the scheduler run 1
- extensible by design, to allow later additions of axiom selections or other methods

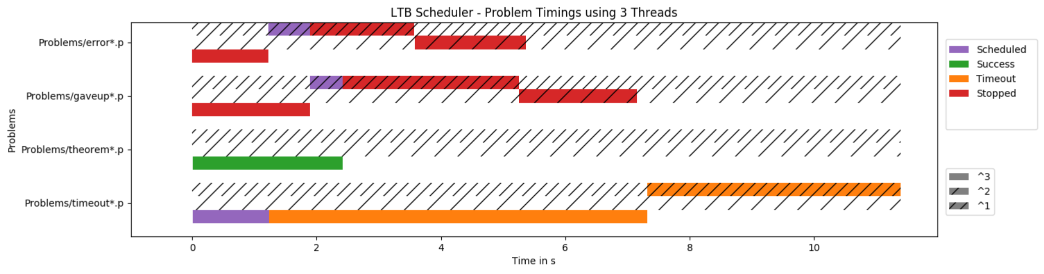


Figure 1: Automatically generated scheduler time-profile-plot using 4 problems with 3 variants (HOL). A box represents a timeinterval of state of a problem during the run of the scheduler.

This removes the boilerplate to implement LTB strategies and allows the implementation, testing and benchmarking of different LTB strategies on a generalized foundation.

Results A first implemention is hosted on github[1]. We will use this implementation at the next CASC at CADE-27 (Aug 25th-30th), first results of the competition should be available at Deduktionstreffen 2019.

[1]https://github.com/leoprover/ltb

References

[HK17] Julio Cesar Lopez Hernandez and Konstantin Korovin. Towards an abstraction-refinement framework for reasoning with large theories. *IWIL@ LPAR*, 1, 2017.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[PSST10] Adam Pease, Geoff Sutcliffe, Nick Siegel, and Steven Trac. Large theory reasoning with sumo at casc. *Ai Communications*, 23(2-3):137–144, 2010.

[SB18] Alexander Steen and Christoph Benzmüller. The higher-order prover leo-iii. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 108–116. Springer, 2018.

[Sut09] Geoff Sutcliffe. The 4th ijcar automated theorem proving system competition–casc-j4. *Ai Communications*, 22(1):59–72, 2009.

[UV13] Josef Urban and Jiří Vyskočil. Theorem proving in large formal mathematics as an emerging ai field. In *Automated Reasoning and Mathematics*, pages 240–257. Springer, 2013.

Computer-Assisted Reasoning about Norms and Obligations

Gavin Goerke

The American University of Paris

1 Introduction

NormativeAI is an open source web application with the purpose of providing an intuitive user interface for both the process of translating a natural language text into formal language and the process of writing queries over the user defined formalizations. NAI is freely available at the URL https://nai.uni.lu and the source code is found at the URL https://github.com/normativeai. The tool is currently being developed to provide the benefits of automated reasoning systems to domains which depend on normative reasoning, particularly the field of law. Such benefits include the added precision that formal logic provides to reasoning as well as the automation of deductions from premises. NAI is being developed to have a practical use case and high-quality user experience and therefore uses the software as a service model wherein no installation of any application is necessary and the tool is easily accessible on a number of devices.

To start using NAI the user logs into the website and pastes the text they wish to formalize into the legislation editor, at this point they can start highlighting text to define terms and relate them using logical connectives. Once this process is completed the user can take advantage of the back-end theorem prover and begin to write queries. The queries allow the user to check the logical consequences, consistency, and independence of the formalization and through this the consequences, consistency, and independence of their particular interpretation of the text. It is worth pausing here to note that the nature any natural language entails a certain amount of ambiguity and so the proofs resulting from the theorem prover are always in the context of the users formalization of their particular interpretation of the text. We can view this as a sort of threefold distinction between formalization, interpretation and the text itself where the user provides the relations between.

We believe that the benefit of NAI is not limited to the results of the queries but that the process of formalization itself provides additional value. In his paper [1], Allen shows how the use of symbolic logic in the process of legal drafting can reduce ambiguity. Similarly, the process of formalizing existing legal texts forces the user notice points of ambiguity and hence the user may be more conscious of the gap between possible interpretations and their own.

2 Logic

The Stanford Encyclopedia of Philosophy describes deontic logic as the branch of symbolic logic that has been most concerned with the contribution of the notions of the permissible, impermissible, obligatory, omissible and ought [3]. This type of logic then is especially well suited for our purposes of automated reasoning over normative concepts. The current iteration of the tool uses an extension of the deontic logic SDL (Standard Deontic Logic) called DL*. While SDL can be encoded in theorem provers without great difficulty it also suffers from a lack of expressivity and an inability to capture common scenarios that appear in legal text such as so-called contrary-to-duty scenarios in which a secondary obligation is only brought into effect if a primary obligation is violated [5]. As shown in [2] extensions such as DL* overcome several of these problems while still being able to be easily encoded in automated theorem provers.

3 Limitations and Planned Features

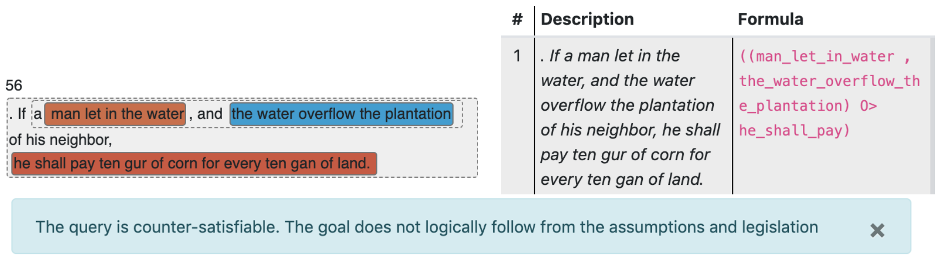
Currently NAI is in an early stage of development however it's core functionality, that is the user interface for the formalization process, the ability to write queries, and the back-end automated theorem prover, are fully functional. In this section we describe some of the limitations as well as possible solutions along with some features with planned implementation in the future.

One limitation is the ability of the current logic to represent defeasible reasoning or reasoning in which the inferences become retractable when some new information is added. Because the current underlying logic is monotonic we do not have the capability to capture this concept and it has been argued (as in [4]) that defeasibility plays an important rule in legal reasoning. A solution to this which is being currently investigated is the implementation of defeasibile reasoning at the application level rather than the logical level, other solutions include using non-monotonic logics as a replacement for or in conjunction with the current logic.

A feature currently being considered is the ability for users to make their annotated texts public. NAI could then build a searchable database of users shared texts with the goal of facilitating discussion and experimentation with formalizations, this public use and discussion would in turn provide valuable feedback to inform the further development of the tool.

4 Using NAI

In this section we give a brief example of annotating a text in NAI. We first login at the home page and are then brought to the dashboard. Under the "Legislations" heading we click "Create new" and are then brought to the "Legislation editor." Here we paste the text we wish to annotate. Once annotated (top left) we can then see the resulting formalization (top right). The bottom image shows the result of a query asking whether a man who let in water to the plantation but the water did not overflow into the plantation of his neighbor, is obliged to pay.



56
. If a man let in the water , and the water overflow the plantation
of his neighbor,
he shall pay ten gur of corn for every ten gan of land.

#	Description	Formula
1	*. If a man let in the water, and the water overflow the plantation of his neighbor, he shall pay ten gur of corn for every ten gan of land.*	`((man_let_in_water , the_water_overflow_the_plantation) O> he_shall_pay)`

The query is counter-satisfiable. The goal does not logically follow from the assumptions and legislation

References

[1] Layman E Allen. 1956. *Symbolic logic: A razor-edged tool for drafting and interpreting legal documents.*. Yale LJ 66 (1956), 833.

[2] Tomer Libal and Matteo Pascucci. 2018. *Automated Reasoning in Normative Detachment Structures with Ideal Conditions.* CoRR abs/1810.09993. arXiv:1810.09993

[3] McNamara, Paul. *Deontic Logic* The Stanford Encyclopedia of Philosophy `https://plato.stanford.edu/archives/sum2019/entries/logic-deontic`

[4] Sartor, Giovanni. 2009. *Defeasibility in Legal Reasoning.* The Logic of Legal Requirements: Essays on Defeasibility.

[5] Prakken, H. and Sergot, M. 1996. *Contrary-to-duty obligations.* Studia Logica 57, 91–115

[6] Tomer Libal and Alexander Steen. 2019. *NAI - The Normative Reasoner.*

Experiments in Deontic Logics using Isabelle/HOL

Ali Farjami

University of Luxembourg, Luxembourg
`ali.farjami@uni.lu`

Deontic logic is a reasoning framework about normative concepts such as obligation, permission, and prohibition. On one hand, we have the family of traditional deontic logics which includes Standard Deontic Logic (SDL),a modal logic of type KD, and Dyadic Deontic Logic (DDL) [5, 6]. On the other hand, we have the so called norm-based deontic logics. Here the frameworks do not evaluate the deontic operators with regard to a set of possible worlds but with reference to a set of norms. Such a framework investigates which norms apply for a given input set, referred to as facts, and a set of explicitly given conditional norms, referred to as normative system. A particular framework that falls within this category, is called Input/Output (I/O) logic. It gained high recognition in the AI community and is also addressed as a chapter in the handbook of deontic logic [6]. The framework is expressive enough for dealing with legal concepts such as constitutive, prescriptive and defensible rules [4].

Current research at the University of Luxembourg focuses on *shallow semantical embeddings* of a family of deontic logics in classical higher-order logic (HOL) [2, 3, 1]. The embeddings have been encoded in Isabelle/HOL, which turns this system into a proof assistant for deontic logic reasoning. The experiments with this environment provide evidence that the shallow semantical embedding methodology fruitfully enables interactive and automated deontic reasoning at the meta-level and the object-level in Isabelle/HOL.

We will present ongoing work on the study of these embeddings and their applications for legal and ethical computerized applications.

References

[1] Christoph Benzmüller, Ali Farjami, Paul Meder, and Xavier Parent. I/O logic in HOL. *Journal of Applied Logics – IfCoLoG Journal of Logics and their Applications*, 2019. To appear, preprint: `https://www.researchgate.net/publication/332786587_IO_Logic_in_HOL`.

[2] Christoph Benzmüller, Ali Farjami, and Xavier Parent. A dyadic deontic logic in HOL. In Jan Broersen, Cleo Condoravdi, Shyam Nair, and Gabriella Pigozzi, editors, *Deontic Logic and Normative Systems — 14th International Conference, DEON 2018, Utrecht, The Netherlands, 3-6 July, 2018*, volume 9706, pages 33–50. College Publications, 2018. John-Jules Meyer Best Paper Award, Preprint: `https://tinyurl.com/y9wp4p6s`.

[3] Christoph Benzmüller, Ali Farjami, and Xavier Parent. Åqvist's dyadic deontic logic E in HOL. *Journal of Applied Logics – IfCoLoG Journal of Logics and their Applications*, 2019. To appear, preprint: `https://www.researchgate.net/publication/332786724_Aqvist's_Dyadic_Deontic_Logic_E_in_HOL`.

[4] Guido Boella and Leendert W. N. van der Torre. Regulative and constitutive norms in normative multiagent systems. In *KR*, pages 255–266. AAAI Press, 2004.

[5] Dov Gabbay, Jeff Horty, Xavier Parent, Ron van der Meyden, and Leon van der Torre. *Handbook of deontic logic and normative systems, Volume I.* College Publication, UK, 2013.

[6] Dov Gabbay, Jeff Horty, Xavier Parent, Ron van der Meyden, and Leon van der Torre. *Handbook of deontic logic and normative systems, Volume II.* College Publication, UK, 2013.

Author Index